Towards Analyzing MongoDB NoSQL Security and Designing Injection Defense Solution

Boyu Hou, Yong Shi, Kai Qian
Department of CS
Kennesaw State University
Marietta, GA, USA
bhou@students.kennesaw.edu,
yshi5,kqian@kennesaw.edu

Lixin Tao
Department of CS
Pace University
New York, USA
ltao@pace.edu

***Abstract*—An increasing number of people and companies have started to use NoSQL database for data management and database design. It has efficient and powerful functions for big data while doing analysis and generalization. In today's world, tons of data need to be stored and executed every second, which possibly brings malicious information into the databases. NoSQL databases, similarly to SQL databases, can be hacked, injected** with malicious codes, or **even destroyed, thus security has been a critical issue for big data analysis using NoSQL databases. In this paper, we analyze the malicious injection in NoSQL databases and propose defense approaches by the utilization of JavaScript and HTTP. MongoDB is one of the most stable and secure NoSQL databases among all NoSQL databases. We demonstrate server-side JavaScript and HTTP injection attacks and propose defense measures to promote the security of MongoDB, which will help NoSQL databases programmers and designers be aware of injection mechanism and build a more secure data environment.**

Keywords—NoSQL Injection, Security Analysis, Defense

I. INTRODUCTION

Data security is becoming more and more important in our world because of big data transactions. Tons of data have been generated, stored, modified, and transferred every second, and databases need to monitor every piece of data in order to maintain a safe environment. However, monitoring every piece of data in databases consumes too much memory, processes, and time, which produces a negative influence upon databases maintenance, slowing the data transactions. One efficient way to solve this problem is to check the data when it enters the database. To facilitate data storage for the users, a system normally provides an input box, which is targeted by the malicious attach. Injection, one of the hacking methods, can be executed to the databases when users input their information. Traditional SQL databases have been used for decades, and with the generation of big amount of data on the daily basis, more and more companies started to choose NoSQL databases for their data storage and transaction. A lot of research has been done on injection and prevention of malicious injection to SQL databases. Similarly, injections can also be executed in NoSQL databases, causing security issues. There are a lot of methods for injection. In this paper, we will focus on two of the injection methods, JavaScript and HTTP trespassing. Meanwhile, by explaining these injections, we will provide the defending solutions.

JavaScript is one of the most popular programming languages in websites and applications, which provides efficient, convenient, and portable functionalities. With JavaScript, programmer and designer can build more functionalities and dynamic views, which produce a positive influence upon website and server. It has been used in many applications to connect NoSQL databases such as MongoDB. MongoDB also provide drives and well-supported command executions, which allows the programmer and designer run JavaScript files exactly in MongoDB server side, in order to make maintenance easier and more efficient. One of the malicious attacks is to close a query statement and inject a piece of malicious code. When the query is executed, the injection will also be executed at the same time. Although the injected query statement still seems correct, it is actually executed twice or even more times because of the malicious injection.

Another kind of injection is HTTP trespassing, which uses the website or server to make injections. When a user goes to websites in a browser, the browser needs to download the source code and show the entire webpage to the user. In order to show the functionalities and effects, the browser needs to execute the source code, which may contain JavaScript, PHP, ASP or others web programming languages. This is where the injection can be executed. If a hacker knows the target MongoDB database name, collection name, port number, user name, and password, he/she can maliciously inject bogus data into the database.

In this paper, we will analyze these two injection methods thoroughly, and propose defense solutions in order to improve and enhance the database security. It is well known that database security is a very comprehensive issue, involved with various issues such as version management, database table management, data access permissions. It is also related to client program management, application security check, client IP control, DBA authority, database audit and data backup. In order to enhance the database security, a security manager need to consider various issues such as intrusion prevention,

978-1-5090-6296-6/17 $31.00 © 2017 IEEE
DOI 10.1109/BigDataSecurity.2017.29

access authentication, and data encryption, in order to reduce the risk of data leakage or tampering we will mainly analyze their mechanisms and demonstrate their constructions.

II. Related Works

NoSQL injection attack tries to inject code when the inputs are not sanitized. The solution is simply to sanitize them before using them [1]. NoSQL database security plays an important role in the data security and network security fields, and scientists have conduct related research work on it. NoSQL databases provide looser consistency restrictions than traditional SQL databases does. By requiring fewer relational constraints and consistency checks, NoSQL databases often offer performance advantages and scaling benefits [2]. One of the attacks is called preparatory step attack, which collects the type and structure of the database to prepare for other types of attacks. It can be injected by using JavaScript and PHP code [3], where hackers inject malicious code into input boxes or directly into the URL. For example, in search functions, the malicious code becomes a variable that participates in execution [4].

III. NOSQL INJECTION

Nowadays, a large amount of data have been generated and transferred worldwide. Critical and sensitive information such as financial account activities and transactions is transferred among computer systems, electronic banking services and mobile devices through communication lines or wirelessly, making it vulnerable to attacks from the hackers. Law enforcement agencies know the people's criminal records from their computers, and doctors use computers to manage medical records. It is very important that information not be transmitted without undue access (unauthorized access). Data security is playing an important role in the big data era. SQL databases and NoSQL databases have been used in many fields, and since NoSQL databases support big data much better, an increasing number of developers and engineers are focusing on how to build a more secure NoSQL database system. Malicious injections can happen in SQL databases, where hackers insert SQL commands into the Web form and enter the domain name or page request query string, ultimately spoofing the server to execute malicious SQL commands. Hackers can also perform malicious injections in NoSQL databases, which produce a negative influence upon information security of NoSQL databases. NoSQL injection attacks can be performed in areas different from traditional SQL injection applications. For example, NoSQL variants can be executed at the application or database level, depending on the NoSQL API and the data model used, where SQL injection is to be performed within the database engine. Normally, a NoSQL injection attack will be executed when the attack string is parsed, evaluated, or connected to a NoSQL API call. In this paper, we use MongoDB, one of the NoSQL databases to show the injection examples and defense approaches.

In general, all kinds of databases have similar functions such as create, retrieve, update and delete. To database users, query could be the most useful function. Query could be demonstrated with an input box or dialog window, where the malicious injection usually happens, because users are allowed to input whatever they want such as notations in those input boxes or dialog windows, which causes serious problems. The query or insert statement might contain some malicious code, and while the original statement is executed, the malicious code is executed as well. One example of the injection methods is "closing the statement", and it will cause problems if the developers do not check and sanitize the user input. Smaller problems include insertion of unexpected data, deletion of target information, and gain of root permission, while larger problems could lead to crashing the whole system or the internet. Notations and segments are the most common methods, which will be further discussed in the following sections with examples and discussion.

A. MongoDB NoSQL Database

MongoDB is a cross-platform document-oriented database which is a type of NoSQL databases. It can be used in applications that process unstructured, semi-structured and polymorphic data, as well as those with large scalability requirements or multi-data center deployments [5]. MongoDB can radically simplify development and operations by delivering a diverse range of capabilities in a single, managed database platform [6]. It supports a very loose data structure similar to the JavaScript Object Notation (JSON) and Binary JSON (BSON) formats, thus it can store complex data types. One of MongoDB's advantages is that it supports a very powerful query language whose syntax is similar to the object-oriented query languages as well as relational database query languages, making it achieve the vast majority of single-function while still supporting the indexing of the data. Like almost every new technology, NoSQL databases lacked security functionalities when they first emerged [7, 8]. In this paper we take MongoDB as an example to show the cases of malicious injection and propose our detection and defense approaches.

MongoDB is used by many commercial projects, especially for those that are involved with big data. MongoDB uses very simple command lines for execution and it can also be imported into many programming languages, and in those programming languages, programmers can use several lines of code to connect to the databases and execute statements. MongoDB is becoming more and more popular for companies and developers because it is efficient, easy to deploy, portable, and dynamic, and it supports internet access and store big data. However, since it is portable, many hackers could hack into MongoDB databases through many programming languages. Usually injection happens in input boxes or dialog windows as we mentioned in previous sections. Besides that, MongoDB also accepts JavaScript files, leading to possible JS file injections.

B. MongoDB NoSQL Database Injection

Hackers leverage existing applications to inject malicious SQL commands into the background database engine, which can be exploited by typing malicious SQL statements into a

website that has security vulnerability. The complexity of detecting SQL injection can best be understood through a variety of examples demonstrating the various SQL injection attack classifications [9].

The first type of method is JavaScript Injection. JavaScript is widely used in a lot of data services because of its convenient and efficient nature, and many companies choose JavaScript to build their websites and servers. JavaScript also supports NoSQL databases very well, and it becomes a positive influence upon big data process. NoSQL databases not only promise simplified development, but also improve security by eliminating the SQL language entirely and relying on much simpler and structured query mechanism that is typically found in the form of JSON and JavaScript [10]. Injection is a common way to hack the databases or even to crash the databases, which would create problems, especially in connected data services. Scnce JavaScript is popularly used, hackers and developers practice injection implementation and defense process on it. One of the injection methods is to input a piece of code in the dialog boxes in order to close the executed statement and implement more. For example, assuming we have a guestbook application that is linked to a MongoDB database and this application contains just one function for testing which handles signing in names. Normally, each guest comes and signs their name into the system, but an injection can insert some unexpected information into the database as well. A sample guestbook is depicted in Figure 1.

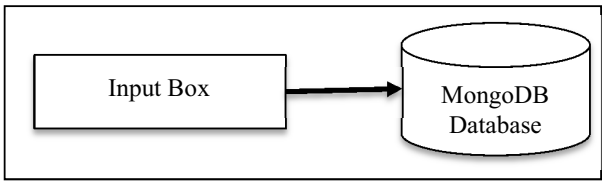

Fig.1 A guestbook diagram for connection between an input box and a MongoDB Database

And its statement is quite simple which is shown in Figure 2.

```
$inset =
"db.getCollection('NameList').insert({'
name':'$param'});";
$response = $db->execute($inset);
```

Fig.2 A query statement to connect to a MongoDB Database

In this case, the database will receive the value from a dialog box by `$param`, and it will be stored into the database. A normal user would input a regular name such as "John" into the dialog box. However, a hacker can insert a much longer and malicious string of information such as "*John'}); db.injection.insert({'Success':'1*". For this case, the entire input can be divided into two parts: *John'});* and *db.injection.insert({'Success':'1*. By this division, two parts can be closed separately by the semicolon, and two statements instead of one will be executed, resulting in (1) the original name being inserted into collection *NameList* and (2) a new collection *injection* being generated, and *Success:1* is able to be written into collection *injection*, which is shown in Figure 3 and Figure 4.

```
MongoDB Enterprise > show collections
Brand
CarBrand
NameList
androidDB
bookinformation
injection
system.indexes
users
MongoDB Enterprise >
```

Fig.3 The list of collection after the malicious injection

```
MongoDB Enterprise > db.injection.find()
{ "_id" : ObjectId("5820e9bee60efa6c9a717447"), "Success" : "1" }
MongoDB Enterprise >
```

Fig.4 The content of collection named "injection" after the malicious injection

This is one of the easiest ways to test the vulnerability of the database system and implement the malicious injection. The hackers would need to know exactly how statements are interpreted and executed in the system in order to design the malicious code to match and close it. Sometimes, the simpler cases are the ones that can be ignored more easily.

The second type of method is JavaScript File Injection. MongoDB allows the developer to load JavaScript files in order to deal with specific piece of data and daily maintenance. By using this method, the developer can build their own files, and it is easy to execute these files in MongoDB. One advantage is that a friendly GUI can be built for the company, and it can be customized. But sometimes this file could contain some malicious code that can be executed and it can conduct some dangerous processes such as read, write or even grab the administration role. This execution is like Figure 5.

```
MongoDB Enterprise > load("injection.js")
true
MongoDB Enterprise >
```

Fig.5 An example to load a JS file in the system

JS file execution brings more efficient process for both designers and users, but if it is not filtered, hackers can use this method to gain something they want. Here we will use the guestbook MongoDB database system again as an example. First of all, the file needs to make a connection to the database, and then it needs to make an insert statement. Because all the code in the file has not been filtered, the malicious code can be executed without validation. For example, the file could contain the insertion of other information into the database which is shown in Figure 6.

```
db =
connect("localhost:27020/example");
db.injection.insert({'Success':'1'});
print("Injection Succeed");
```

Fig.6 An example of a JS file that contains malicious information

In the example shown in Fig. 6, the local server, port number and database name *`localhost:27020/example`* are registered and an insert statement *`db.injection.insert({'Success':'1'});`* is also injected. The result is shown in Fig.7.

```
MongoDB Enterprise > load("injection.js")
connecting to: localhost:27017/example
Injection Succeed
true
```

Fig.7 Testing result of loading a JS file in the database system

The third type of method is HTTP trespassing, which uses website source code for malicious injections. Databases in the companies stay in their own systems which are protected by the company firewalls, thus it is very hard to hack into them from the outside. When a user visits a website, the user's computer needs to download all the source code from the website in order for the browser to translate and execute so that the user can see words and pictrues and use functions. The source usually is not checked and filtered, and it may contain some code that does not affect the view or function, however, it may still be dangerous.

HTTP trespassing is utilized to bypass the firewall, and in order to do so; it needs someone inside the firewall to visit the target website which contains malicious code. When this person visits that website, the source code will be downloaded to the system inside the firewall, and all of the code can be executed completely and unconsciously. The diagram is shown in Figure 8.

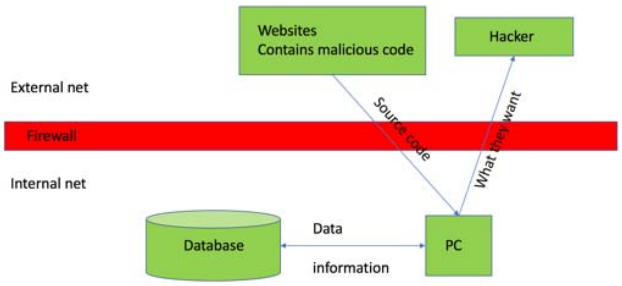

Fig.8 HTTP trespassing diagram

The next step is to know the target database address, port number and collection name. The target website that contains malicious code can be the site can be written in many ways, and the languages support functions can be selected. Here we choose JavaScript as an example. The code written in JavaScript is shown in Figure 9.

```
<script>
    var MongoClient =
require('mongodb').MongoClient;
MongoClient.connect("mongodb://10.0.
0.30:27017/example", function(err,
db) {
     if(!err) {
        console.log("We are
connected");
     }
     var collection =
db.collection('injection');
     var doc1 = {'Success':'1'};
     collection.insert(doc1);
          });
</script>
```

Fig.9 JavaScript injection code

In the example shown in Fig. 9, the target database is deployed in the address of 10.0.0.30. It then creates a collection called injection and inserts a piece of data: *Success:1*.

This JavaScript can be embedded into many websites and it can be executed while someone is browsing the site. The result is shown in Fig.10.

```
MongoDB Enterprise > db.injection.find()
{ "_id" : ObjectId("5820e9bee60efa6c9a717447"), "Success" : "1" }
MongoDB Enterprise >
```

Fig.10 trespassing result

Basically, in order to make successful malicious injections, the hackers need to perform the following steps. First of all, the hackers need to make sure that the condition statement can be executed while using such as query function so that the malicious code will be processed and go through the entire database to acquire what the hackers want. The second step is to try to build the statement in order to get the higher level permissions such as root or admin permission which will produce an extremely negative influence upon personal or company's data security. The third step is to try to avoid firewall and take advantage of the leak of checking and filtering in order to let malicious code be downloaded into the target system.

IV. MONGODB NOSQL DEFENSE AND DETECTION ANALYSIS

When it comes to security issues, no matter how much effort we make our systems robust, as long as there is one place of negligence, the attackers will use this point to make a breakthrough, and all the effort we put is in vain. The dangers of injection make enterprises have to pursue a well-developed defense strategy, however, thorough and successful defense can only be achieved little by little over the time, and it is

more of a systematic and management problem rather than a technical one.

Malicious injection attacks can cause a lot of problems: the database server can be remotely controlled and attacked, a back door can be installed, and the database administrator account can be tampered with. Furthermore, since there is a possibility to access operating systems through database servers, the hackers can modify or control the operating systems as well, and destroy the hard disk data and paralyze the entire system. Therefore, it is necessary to build a defense strategy in order to improve the system security. Defence can be demonstrated in input parts and background parts, and we will provide detection approaches for different cases in the following subsections.

A. MongoDB NoSQL Defense Analysis

In this paper, we will discuss three database defense solutions. The first one input validation which is to limit the input of the users. For example, for the guestbook database system, when developers build the system and software, they can add regular expression code to limit the input box to only accept letters. An example of the regular expression code is shown in Figure 11.

```
String NAME_PATTERN = "[a-z\\sA-Z]+";
        Pattern        pattern        =
Pattern.compile(NAME_PATTERN);
        Matcher        matcher        =
pattern.matcher(name);
        return matcher.matches();
```

Fig. 11 Regular expression code to limit user input

Regular expression is efficient in preventing all unwanted notations or characters from getting inputted into the database. Since almost all injection methods contain notations, the malicious code cannot pass the validation process provided by this solution.

The second defense solution is to assign the permission to all users so that it can prevent the JS file injecton. At the early stage, NoSQL did not support proper authentication and role management [11], although the vast majority of Mongo applications always have varying degrees of security requirements. In general, applications need to ensure that they meet several basic security requirements. First of all, the applications must be able to identify the visitor. Secondly, the applications must provide security guarantees against unauthorized access to the database resources. Finally, in a multi-tier application model, the applications need to have logical components that can provide protection against illegal user calls from bypassing the database.

In a department of a company, the roles can be divided into managers, employees and other categories. Each user obtains the appropriate permissions according to their role, which greatly simplifies the authorization logic. For example, in a payroll system, employees can check their own salary information, however, they do not have the right to change their own pay. On the other hand, managers can adjust the wages of staff, and administrators can create the wage for new employees and have also regular backup data system maintenance authority. If a wage system user is promoted from employee to manager, the user's role in the wage system changes from employee to manager, thereby gaining manager-level privileges. Therefore, in the role-based system, the authorization is not directly linked with the user, but linked to the role. The user, through the role of association, can ultimately get the appropriate authorization. There are different levels of permissions. For example, operation authority is the restraint of the operation of the system, mainly for WEB page initiated request for permission filtering which is based on the URL. Another example is data permission which is for specific data entities. It describes the database row-level operating authority including increase, delete, change, and check operation. Each operation corresponds to a fine-grained data permission, as well as the role of the function.

The third defense solution is to check and filter the variables of a statement. As the dynamic SQL statement discussed in the previous sections shows, malicious SQL code is “stitched” into the SQL statement which will be the implementation of the database. This solution is to check the parameterized statement before stitching it. At the time of writing condition statements, this solution prohibits the variable for user input to be directly and automatically embedded into the condition statement. Instead, the contents of the user's input must be filtered, using parameterized statements (a feature used to execute the same or similar database statements repeatedly with high efficiency) to pass variables entered by the user. Parameterized statements uses parameters instead of embedding user input variable into the condition statement. With this measure, most of injection attacks can be eliminated. If a user intends to pass $_GET/$_POST parameters to a query, the user needs to make sure that the parameters are cast to strings first [12]. For example, there is a piece of code to determine the characters in a variable if it contains numbers only or not. By checking the names, if yes, pass the value; if not, reject the value. The simple code is shown in Figure 12.

```
String            m_argv[]            =
{input.getText().toString()};
```

Fig. 12 Parameterized statement code

Based on the discussion above, it is very important to build a comprehensive permission system for MongoDB and users, and assigning permission levels to all users makes a more secure database.

B. Malicious Feature Detection

In the last few years, security experts have paid more attention to attacks at the network application layer. This is because no matter how strong the firewall rules or diligent patchwork patching mechanism is, if web application developers do not follow the security code for development,

attackers can still enter the system through the port. Malicious feature detection approach detects if the system or software has some features which are dangerous for security. Based on malicious code and features, the chart of patterns is shown below:

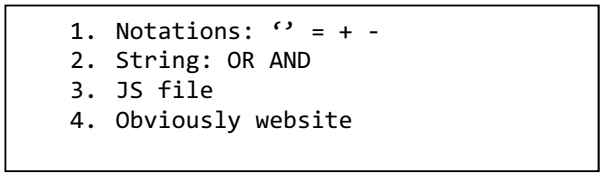

Fig. 13 defense detection chart

This simple detection chart can help developers detect malicious code or injected statement in their applications and systems. This provides different kinds of regular expression rules to detect SQL injection and cross-site scripting attacks. These simple rules are modified to take advantage of additional styles and types so that they can detect malicious code more accurately. It is recommended that developers use these rules as debugging IDS or log analysis of the starting point when they develop network applications for attack detection. After several revisions, developers will be ready to detect those attacks after evaluating the non-malicious responses in the normal network transactions section.

V. CONCLUSION AND FUTURE WORK

Information security plays a significant role in systems and software, which requires developers and supervisors pay extra attention to database, permission, and firewalls. Most of the databases are not entirely invulnerable. There are always vulnerabilities that could be attacked by hackers for malicious purposes. The malicious attack cases discussed in this paper are just a few of them. Information can be easily leaked, intercepted, and tampered. A single security measure will not be able to ensure the safety of communications, and information must be integrated with various security measures through technical, administrative and administrative means, in order to achieve the purpose of confidential information security.

In the future work, we will continue focusing on NoSQL databases. We will analyze more types of malicious NoSQL database attacks, as well as propose detection solutions to prevent them, which will build a more secure information databases.

REFERENCES

[1] "NoSQL Injection in MongoDB" https://zanon.io/posts/nosql-injection-in-mongodb

[2] https://www.owasp.org/index.php/Testing_for_NoSQL_injection

[3] http://software-talk.org/blog/2015/02/mongodb-nosql-injection-security/

[4] Boyu, Hou, et al. “MongoDB NoSQL Injection Security Threat Analysis and Detection”

[5] http://blog.securelayer7.net/mongodb-security-injection-attacks-with-php/

[6] https://marketplace.nyc/p/mongodb-atlas/

[7] No SQL and No Security https://www.securosis.com/blog/nosql-and-nosecurity

[8] Okman, Lior, et al. "Security issues in nosql databases." Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on. IEEE, 2011.

[9] http://www.dbnetworks.com/pdf/sql-injection-detection-web-environment.pdf

[10] http://blog.websecurify.com/2014/08/hacking-nodejs-and-mongodb.html

[11] Factor, Michael, et al. "Secure Logical Isolation for Multi-tenancy in cloud storage." Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on. IEEE, 2013.

[12] http://php.net/manual/en/mongo.security.php